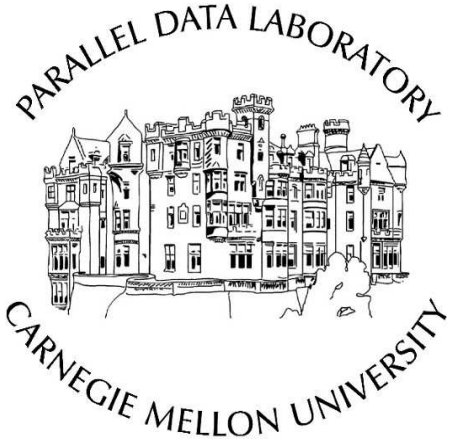


A (In)Cast of Thousands: Scaling Datacenter TCP to Kiloservers and Gigabits

Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat,
David G. Andersen, Gregory R. Ganger, Garth A. Gibson
Carnegie Mellon University

CMU-PDL-09-101

February 2009

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

This paper presents a practical solution to the problem of high-fan-in, high-bandwidth synchronized TCP workloads in datacenter Ethernets—the Incast problem. In these networks, receivers often experience a drastic reduction in throughput when simultaneously requesting data from many servers using TCP. Inbound data overfills small switch buffers, leading to TCP timeouts lasting hundreds of milliseconds. For many datacenter workloads that have a synchronization requirement (e.g., filesystem reads and parallel data-intensive queries), incast can reduce throughput by up to 90%.

Our solution for incast uses high-resolution timers in TCP to allow for microsecond-granularity timeouts. We show that this technique is effective *in avoiding incast using simulation and real-world experiments. Last, we show that eliminating the minimum retransmission timeout bound is* safe *for all environments, including the wide-area.*

Acknowledgements: We would like to thank Brian Mueller at Panasas Inc. for helping us to conduct experiments on their systems. We also would like to thank our partners in the Petascale Data Storage Institute, including Andrew Shewmaker, HB Chen, Parks Fields, Gary Grider, Ben McClelland, and James Nunez at Los Alamos National Lab for help with obtaining packet header traces. We thank the members and companies of the PDL Consortium (including APC, Cisco, Data Domain, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, NetApp, Oracle, Seagate, Sun Microsystems, Symantec, and VMware) for their interest, insights, feedback, and support. Finally, we'd like to thank Michael Stroucken for his help managing the PDL cluster. This material is based on research sponsored in part by the National Science Foundation, via grants #CNS-0546551, #CNS-0326453 and #CCF-0621499, by the Army Research Office under agreement number DAAD19–02–1–0389, by the Department of Energy under Award Number #DE-FC02-06ER25767, and by DARPA under grant #HR00110710025.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
FEB 2009		**00-00-2009 to 00-00-2009**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
A (In)Cast of Thousands: Scaling Datacenter TCP to Kiloservers and Gigabits	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,Parallel Data Laboratory,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**19**	

Keywords: Cluster-based storage systems, TCP, performance measurement and analysis

1 Introduction

In its 35 year history, TCP has been repeatedly challenged to adapt to new environments and technology. Researchers have proved adroit in doing so, enabling TCP to function well in gigabit networks [27], long/fat networks [18, 10], satellite and wireless environments [22, 7], among others. In this paper, we examine and improve TCP's performance in an area that, surprisingly, proves challenging to TCP: very low delay, high throughput, datacenter networks of dozens to hundreds of machines.

The problem we study is termed *incast*: A drastic reduction in throughput when multiple senders communicate with a single receiver in these networks. The highly bursty, very fast data overfills typically small Ethernet switch buffers, causing intense packet loss that leads to TCP timeouts. These timeouts last hundreds of milliseconds on a network whose round-trip-time (RTT) is measured in the 10s or 100s of microseconds. Protocols that have some form of synchronization requirement—filesystem reads, parallel data-intensive queries—block waiting for the timed-out connections to finish. These timeouts and the resulting delay can reduce throughput by 90% (Figure 1, 200ms RTO_{min}) or more [25, 28].

In this paper, we present and evaluate a set of system extensions to enable microsecond-granularity timeouts – the TCP retransmission timeout (RTO). The challenges in doing so are threefold: First, we show that the solution is *practical* by modifying the Linux TCP implementation to use high-resolution kernel timers. Second, we show that these modifications are *effective*, enabling a network testbed experiencing incast to maintain maximum throughput for up to 47 concurrent senders, the testbed's maximum size (Figure 1, 200μs RTO_{min}). Microsecond granularity timeouts are necessary—simply reducing RTO_{min} to 1ms without also improving the timing granularity may not prevent incast. In simulation, our changes to TCP prevent incast for up to 2048 concurrent senders on 10 gigabit Ethernet. Lastly, we show that the solution is *safe*, examining the effects of this aggressively reduced RTO in the wide-area Internet, showing that its benefits to incast recovery have no drawbacks on performance for bulk flows.

The motivation for solving this problem is the increasing interest in using Ethernet and TCP for inter-

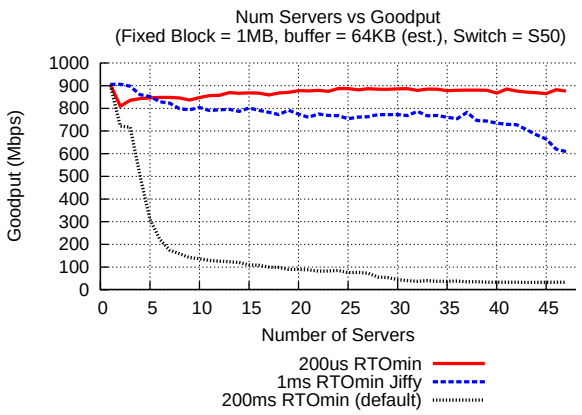


Figure 1: TCP Incast: A throughput collapse can occur with increased numbers of concurrent senders in a synchronized request. Reducing RTO to microsecond granularity alleviates Incast.

processor communication and bulk storage transfer applications in the fastest, largest data centers, instead of Fibrechannel or Infiniband. Provided that TCP adequately supports high bandwidth, low latency, synchronized and parallel applications, there is a strong desire to "wire-once" and reuse the mature, well-understood transport protocols that are so familiar in lower bandwidth networks.

2 Background

Cost pressures increasingly drive datacenters to adopt commodity components, and often low-cost implementations of such. An increasing number of clusters are being built with off-the-shelf rack-mount servers interconnected by Ethernet switches. While the adage "you get what you pay for" still holds true, entry-level gigabit Ethernet switches today operate at full data rates, switching upwards of 50 million packets per second—at a cost of about $10 per port. Commodity 10Gbps Ethernet is now cost-competitive with specialized interconnects such as Infiniband and FibreChannel, and also benefits from wide brand recognition to boot. To reduce cost, however, switches often sacrifice expensive, power-hungry SRAM packet buffers, the effect of which we explore throughout this work.

The desire for commodity parts extends to transport protocols. TCP provides a kitchen sink of pro-

tocol features, giving reliability and retransmission, congestion and flow control, and delivering packets in-order to the receiver. While not all applications need all of these features [20, 31] or benefit from more rich transport abstractions [15], TCP is mature and well-understood by developers, leaving it the transport protocol of choice even in many high-performance environments.

Without link-level flow control, TCP is solely responsible for coping with and avoiding packet loss in the (often small) Ethernet switch egress buffers. Unfortunately, the workload we examine has three features that challenge (and nearly cripple) TCP's performance: a highly parallel, synchronized request workload; buffers much smaller than the bandwidth-delay product of the network; and low latency that results in TCP having windows of only a few packets.

2.1 The Incast Problem

Synchronized request workloads are becoming increasingly common in today's commodity clusters. Examples include parallel reads/writes in cluster filesystems such as Lustre [8], Panasas [34], or NFSv4.1 [33]; search queries sent to dozens of nodes, with results returned to be sorted[1]; or parallel databases that harness multiple back-end nodes to process parts of queries.

In a clustered file system, for example, a client application requests a data block striped across several storage servers, issuing the next data block request only when all servers have responded with their portion. This *synchronized request* workload can result in packets overfilling the buffers on the client's port on the switch, resulting in many losses. Under severe packet loss, TCP can experience a timeout that lasts a minimum of 200ms, determined by the TCP minimum retransmission timeout (RTO_{min}). While operating systems use a default value today that may suffice for the wide-area, datacenters and SANs have round-trip-times that are orders of magnitude below the RTO_{min} defaults:

Scenario	RTT	OS	TCP RTO_{min}
WAN	100ms	Linux	200ms
Datacenter	<1ms	BSD	200ms
SAN	<0.1ms	Solaris	400ms

When a server involved in a synchronized request experiences a timeout, other servers can finish sending their responses, but the client must wait a minimum of 200ms before receiving the remaining parts of the response, during which the client's link may be completely idle. The resulting throughput seen by the application may be as low as 1-10% of the client's bandwidth capacity, and the per-request latency will be higher than 200ms.

This phenomenon was first termed "Incast" and described by Nagle et. al [25] in the context of parallel filesystems. Nagle et. al coped with Incast in the parallel filesystem with application specific mechanisms. Specifically, Panasas [25] limits the number of servers simultaneously sending to one client to about 10 by judicious choice of the file striping policies. It also reduces the default size of its per-flow TCP receive buffers (capping the advertised window size) on the client to avoid incast on switches with small buffers. For switches with large buffers, Panasas provides a mount option to increase the client's receive buffer size. In contrast, this work provides a TCP-level solution to incast for switches with small buffers and many more than 10 simultaneous senders.

Without application-specific techniques, the general problem of incast remains: Figure 1 shows the throughput of our test synchronized-read application (Section 4) as we increase the number of nodes it reads from, using an unmodified Linux TCP (200ms RTO_{min} line). This application performs synchronized reads of 1MB blocks of data; that is, each of N servers responds to a block read request with 1 MB / N bytes at the same time. Even using a high-performance switch (with its default settings), the throughput drops drastically as the number of servers increases, achieving a shockingly poor 3% of the network capacity—about 30Mbps—when it tries to stripe the blocks across all 47 servers.

Prior work characterizing TCP Incast ended on a somewhat down note, finding that existing TCP improvements—NewReno, SACK [22], RED [14], ECN [30], Limited Transmit [3], and modifica-

[1] In fact, engineers at Facebook recently rewrote the middle-tier caching software they use—memcached [13]—to use UDP so that they could "implement application-level flow control for ... gets of hundreds of keys in parallel" [12]

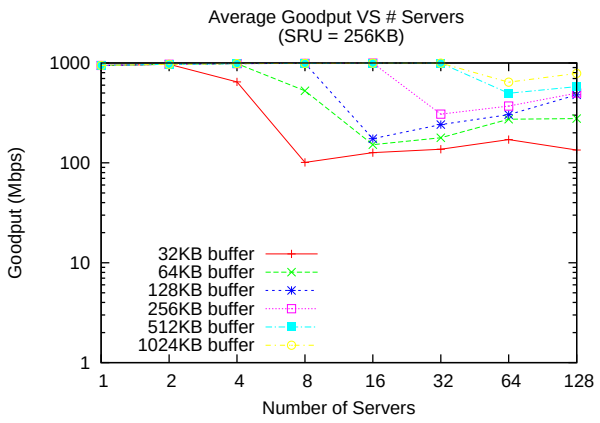


Figure 2: Doubling the switch egress buffer size doubles the numbers of concurrent senders needed to see incast.

tions to Slow Start—sometimes increased throughput, but did not substantially change the incast-induced throughput collapse [28]. This work found three partial solutions: First, as shown in Figure 2 (from [28]), larger switch buffers can delay the onset of Incast (doubling the buffer size doubled the number of servers that could be contacted). But increased switch buffering comes at a substantial dollar cost – switches with 1MB packet buffering per port may cost as much as $500,000. Second, Ethernet flow control was effective when the machines were on a single switch, but was dangerous across inter-switch trunks because of head-of-line blocking. Finally, reducing TCP's minimum RTO, in simulation, appeared to allow nodes to maintain high throughput with several times as many nodes—but it was left unexplored whether and to what degree these benefits could be achieved in practice, meeting the three criteria we presented above: practicality, effectiveness, and safety. In this paper, we answer these questions in depth to derive an effective solution for practical, high-fan-in datacenter Ethernet communication.

3 Challenges to Fine-Grained TCP Timeouts

Successfully using an aggressive TCP retransmit timer requires first addressing the issue of safety and generality: is an aggressive timeout appropriate for use in the wide-area, or should it be limited to the datacenter? Does it risk increased congestion or decreased throughput because of spurious (incorrect) timeouts? Second, it requires addressing implementability: TCP implementations typically use a coarse-grained timer that provides timeout support with very low overhead. Providing tighter TCP timeouts requires not only reducing or eliminating RTO_{min}, but also supporting fine-grained RTT measurements and kernel timers. Finally, if one makes TCP timeouts more fine-grained, how low must one go to achieve high throughput with the smallest additional overhead? And to how many nodes does this solution scale?

3.1 Jacobson RTO Estimation

The standard RTO estimator [17] tracks a smoothed estimate of the round-trip time, and sets the timeout to this RTT estimate plus four times the linear deviation—roughly speaking, a value that lies outside four standard deviations from the mean:

$$RTO = SRTT + (4 \times RTTVAR) \quad (1)$$

Two factors set lower bounds on the value that the RTO can achieve: an explicit configuration parameter, RTO_{min}, and the implicit effects of the granularity with which RTTs are measured and with which the kernel sets and checks timers. As noted earlier, common values for RTO_{min} are 200ms, and most implementations track RTTs and timers at a granularity of 1ms or larger.

Because RTT estimates are difficult to collect during loss and timeouts, a second safety mechanism controls timeout behavior: exponential backoff. After each timeout, the RTO value is doubled, helping to ensure that a single RTO set too low cannot cause an long-lasting chain of retransmissions.

3.2 Is it safe to disregard RTO_{min}?

There are two possible complications of permitting much smaller RTO values: spurious (incorrect) timeouts when the network RTT suddenly jumps, and breaking the relationship between the delayed acknowledgement timer and the RTO values.

Spurious retransmissions: The most prominent study of TCP retransmission showed that a high (by the standards of datacenter RTTs) RTO_{min}

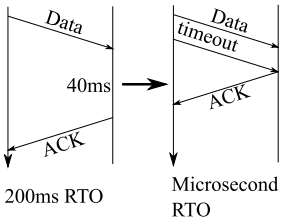
helped avoid spurious retransmission in wide-area TCP transfers [4], regardless of how good an estimator one used based on historical RTT information. Intuition for why this is the case comes from prior [24, 11] and subsequent [35] studies of Internet delay changes. While most of the time, end-to-end delay can be modeled as random samples from some distribution (and therefore, can be predicted by an RTO estimator), the delay consistently observes both occasional, unpredictable delay spikes, as well as shifts in the distribution from which the delay is drawn. Such changes can be due to the sudden introduction of cross-traffic, routing changes, or failures. As a result, wide-area "packet delays [are] not mathematically [or] operationally steady" [35], which confirms the Allman and Paxson observation that RTO estimation involves a fundamental tradeoff between rapid retransmission and spurious retransmissions.

Fortunately, TCP timeouts and spurious timeouts were and remain rare events, as we explore further in Section 8. In the ten years since the Allman and Paxson study, TCP variants that more effectively recover from loss have been increasingly adopted [23]. By 2005, for example, nearly 65% of Web servers supported SACK [22], which was introduced only in 1996. In just three years from 2001—2004, the number of TCP Tahoe-based servers dropped drastically in favor of NewReno-style servers.

Moreover, algorithms to undo the effects of spurious timeouts have been both proposed [4, 21, 32] and, in the case of F-RTO, adopted in the latest Linux implementations. The default F-RTO settings conservatively halve the congestion window when a spurious timeout is detected but remain in congestion avoidance mode, thus avoiding the slow-start phase. Given these improvements, we believe disregarding RTO_{min} is safer today than 10 years ago, and Section 8 will show measurements reinforcing this.

Delayed Acknowledgements: The TCP delayed ACK mechanism attempts to reduce the amount of ACK traffic by having a receiver acknowledge only every other packet [9]. If a single packet is received with none following, the receiver will wait up to the delayed ACK timeout threshold before sending an ACK. The delayed ACK threshold must be shorter than the lowest RTO values, or a sender might time out waiting for an ACK that is merely delayed by the receiver. Modern systems set the delayed ACK timeout to 40ms, with RTO_{min} set to 200ms.

Consequently, a host modified to reduce the RTO_{min} below 40ms would periodically experience an unnecessary timeout when communicating with unmodified hosts, specifically when the RTT is below 40ms (e.g., in the datacenter and for short flows on the wide-area). As we show in the following section, delayed ACKs themselves impair performance in incast environments: we suggest disabling them entirely. This solution requires client participation, however, and so is not general.



In the following section, we discuss three practical solutions for interoperability with unmodified, delayed-ACK-enabled clients.

As a consequence, there are few a-priori reasons to believe that eliminating RTO_{min}—basing timeouts solely upon the Jacobson estimator and exponential backoff—would greatly harm wide-area performance and datacenter environments with clients using delayed ACK. We evaluate these questions experimentally in Sections 4 and 8.

4 Evaluating Throughput with Fine-Grained RTO

How low must the RTO be allowed to go to retain high throughput under incast-producing conditions, and to how many servers does this solution scale? We explore this question using real-world measurements and ns-2 simulations [26], finding that to be maximally effective, the timers must operate on a granularity close to the RTT of the network—hundreds of microseconds or less.

Test application: Striped requests. The test client issues a request for a block of data that is striped across N servers (the "stripe width"). Each server responds with $\frac{blocksize}{N}$ bytes of data. Only after it receives the full response from every server will the client issue requests for the subsequent data block. This design mimics the request patterns found in several cluster filesystems and several parallel

workloads. Observe that as the number of servers increases, the amount of data requested from each server decreases. We run each experiment for 200 data block transfers to observe steady-state performance, calculating the goodput (application throughput) over the entire duration of the transfer.

We select the block size (1MB) based upon read sizes common in several distributed filesystems, such as GFS [16] and PanFS [34], which observe workloads that read on the order of a few kilobytes to a few megabytes at a time. Prior work suggests that the block size shifts the onset of incast (doubling the block size doubles the number of servers before collapse), but does not substantially change the system's behavior; different systems have their own "natural" block sizes. The mechanisms we develop improve throughput under incast conditions for any choice of block sizes and buffer sizes.

In Simulation: We simulate one client and multiple servers connected through a single switch where round-trip-times under low load are 100μs. Each node has 1Gbps capacity, and we configure the switch buffers with 32KB of output buffer space per port, a size chosen based on statistics from commodity 1Gbps Ethernet switches. Because ns-2 is an event-based simulation, the timer granularity is infinite, hence we investigate the effect of RTO_{min} to understand how low the RTO needs to be to avoid Incast throughput collapse. Additionally, we add a small random timer scheduling delay of up to 20μs to more accurately model real-world scheduling variance.[2]

Figure 3 depicts throughput as a function of the RTO_{min} for stripe widths between 4 and 128 servers. Throughput using the default 200ms RTO_{min} drops by nearly an order of magnitude with 8 concurrent senders, and by nearly two orders of magnitude when data is striped across 64 and 128 servers.

Reducing the RTO_{min} to 1ms is effective for 8-16 concurrent senders, fully utilizing the client's link, but begins to suffer when data is striped across a larger number of servers: 128 concurrent senders utilize only 50% of the available link bandwidth even

[2]Experience with ns-2 showed that without introducing this delay, we saw simultaneous retransmissions from many servers all within a few microseconds, which is rare in real world settings.

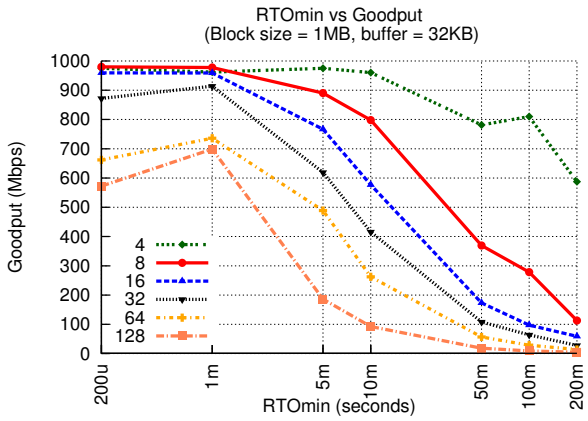


Figure 3: Reducing the RTO_{min} in simulation to microseconds from the current default value of 200ms improves goodput.

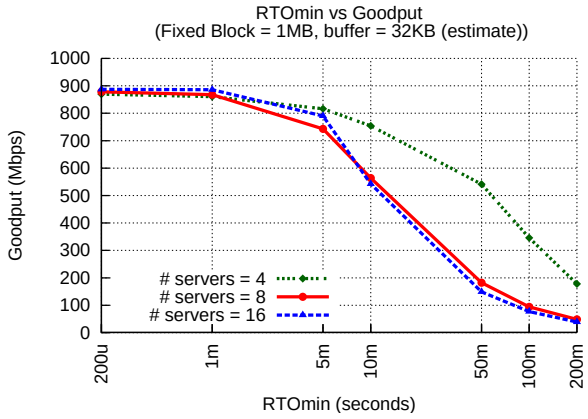


Figure 4: Experiments on a real cluster validate the simulation result that reducing the RTO_{min} to microseconds improves goodput.

with a 1ms RTO_{min}. For 64 and 128 servers and low RTO_{min} values, we note that each individual flow does not have enough data to send to saturate the link, but also that performance for 200μs is worse than for 1ms; we address this issue in more detail when scaling to hundreds of concurrent senders in Section 7.

In Real Clusters: We evaluate incast on two clusters; one sixteen-node cluster using an HP Procurve 2848 switch, and one 48-node cluster using a Force10 S50 switch. In these clusters, every node has 1 Gbps links and a client-to-server RTT of approximately 100μs. All nodes run Linux kernel 2.6.28.

We run the same synchronized read workload as in simulation.

For these experiments, we use our modified Linux 2.6.28 kernel that uses microsecond-accurate timers with microsecond-granularity RTT estimation (§6) to be able to accurately set the RTO_{min} to a desired value. Without these modifications, the default TCP timer granularity in Linux can be reduced only to 1ms. As we show later, when added to the 4x RTTVAR estimate, the 1ms timer granularity effectively raises the minimum RTO over 5ms.

Figure 4 plots the application goodput as a function of the RTO_{min} for 4, 8, and 16 concurrent senders. For all configurations, goodput drops with increasing RTO_{min} above 1ms. For 8 and 16 concurrent senders, the default RTO_{min} of 200ms results in nearly 2 orders of magnitude drop in throughput.

The real world results deviate from the simulation results in a few minor ways. First, the maximum achieved throughput in simulation nears 1Gbps, whereas the maximum achieved in the real world is 900Mbps. Simulation throughput is always higher because simulated nodes are infinitely fast, whereas real-world nodes are subject to myriad influences, including OS scheduling and Ethernet or switch timing differences, resulting in real-world results slightly below that of simulation.

Second, real world results show negligible difference between 8 and 16 servers, while the differences are more pronounced in simulation. We attribute this to variances in the buffering between simulation and the real world. As we show in Section 7, small microsecond differences in retransmission scheduling can lead to improved goodput; these differences exist in the real world but are not modeled as accurately in simulation.

Third, the real world results show identical performance for RTO_{min} values of 200μs and 1ms, whereas there are slight differences in simulation. We find that the RTT and variance seen in the real world is higher than that seen in simulation. Figure 5 shows the distribution of round-trip-times during an Incast workload. While the baseline RTTs can be between 50-100μs, increased congestion causes RTTs to rise to 400μs on average with spikes as high as 1ms. Hence, the variance of the RTO combined with the higher RTTs mean that the actual retransmission timers set by the kernel are between 1-3ms, where an RTO_{min} will show no improvement below 1ms. Hence, where we specify a RTO_{min} of 200μs, we are effectively eliminating the RTO_{min}, allowing RTO to be as low as calculated.

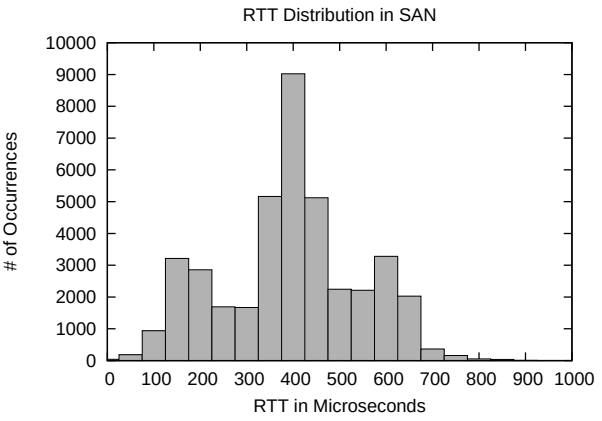
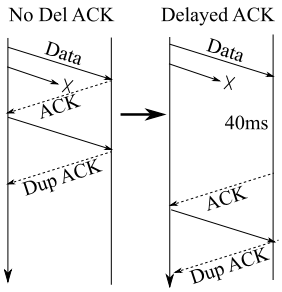


Figure 5: During an incast experiment on a 16-node cluster, RTTs increase by 4 times the baseline RTT (100μs) on average with spikes as high as 1ms. This produces RTO values in the range of 1-3ms, resulting in an RTO_{min} of 1ms being as effective as 200μs in practice.

Despite these differences, the real world results show the need to reduce the RTO to at least 1ms to avoid throughput degradation.

4.1 Interaction with Delayed ACK for Unmodified Clients

During the onset of Incast using five or fewer servers, the delayed ACK mechanism acted as a miniature timeout, resulting in reduced, but not catastrophically low, throughput [28] during certain loss patterns. As shown to the right, delayed ACK can delay the receipt of enough duplicate acks for data-driven loss recovery when the window is small (or reduce the number of duplicate ACKs by one, turning a recoverable loss into a timeout). While this delay is not as high as a full 200ms RTO, 40ms is still large compared to the RTT and results in low



throughput for three to five concurrent senders. Beyond five senders, high packet loss creates 200ms retransmission timeouts which mask the impact of delayed ACK delays.

While servers require modifications to the TCP stack to enable microsecond-resolution retransmission timeouts, the clients issuing the requests do not necessarily need to be modified. But because delayed ACK is implemented at the receiver, a server may normally wait 40ms for an unacked segment that successfully arrived at the receiver. For servers using a reduced RTO in a datacenter environment, the server's retransmission timer may expire long before the unmodified client's 40ms delayed ACK timer fires. As a result, the server will timeout and resend the unacked packet, cutting ssthresh in half and rediscovering link capacity using slow-start. Because the client acknowledges the retransmitted segment immediately, the server does not observe a coarse-grained 40ms delay, only an unnecessary timeout.

Figure 6 shows the performance difference between our modified client with delayed ACK disabled, delayed ACK enabled with a 200μs timer, and a standard kernel with delayed ACK enabled.

Beyond 8 servers, our modified client with delayed ACK enabled receives 15-30Mbps lower throughput compared to the client with delayed ACK disabled, whereas the standard client experiences between 100 and 200Mbps lower throughput. When the client delays an ACK, the standard client forces the servers to timeout, yielding much worse performance. In contrast, the 200μs minimum delayed ACK timeout client delays a server by roughly a round-trip-time and does not force the server to timeout, so the performance hit is much smaller.

Delayed ACK can provide benefits where the ACK path is congested [6], but in the datacenter environment, we believe that delayed ACK should be disabled; most high-performance applications favor quick response over an additional ACK-processing overhead and are typically equally provisioned for both directions of traffic. Our evaluations in Section 6 disable delayed ACK on the client for this reason. While these results show that for full performance, delayed ACK should be disabled, we note that unmodified clients still achieve good performance and avoid Incast.

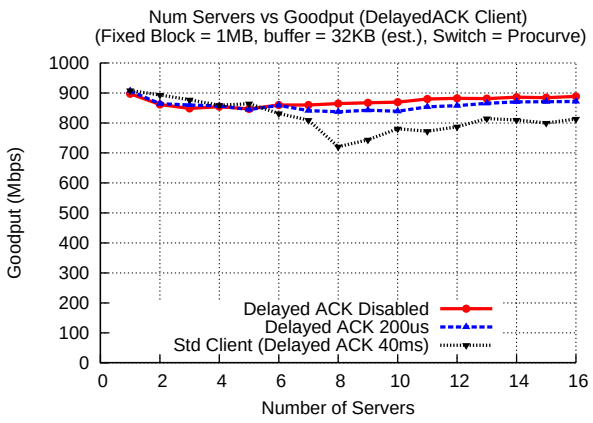


Figure 6: Disabling Delayed ACK on client nodes provides optimal goodput.

5 Preventing Incast

In this section we briefly outline the three components of our solution to incast, the necessity for and effectiveness of which we evaluate in the following section.

1. Microsecond-granularity timeouts. We first modify the kernel to measure RTTs in microseconds, changing both the infrastructure used for timing and the values placed in the TCP timestamp option. The kernel changes use the high-resolution, hardware-supported timing hardware present in modern operating systems. In Section 9, we briefly discuss alternatives for legacy hardware.

2. Randomized timeout delay. Second, we modify the microsecond-granularity RTO values to include an additional randomized component:

$$\text{timeout} = RTO + (rand(0.5) * RTO)$$

In expectation, this increases the duration of the RTO by up to 25%, but more importantly, has the effect of de-synchronizing packet retransmissions. As we show in the next section, this decision is unimportant with smaller numbers of servers, but becomes critical when the number of servers is large enough that a single batch of retransmitted packets themselves cause extensive congestion. The extra half-RTO allows some packets to get through the switch (the RTO is at least one RTT, so half an RTO is

at least the one-way transit time) before being clobbered by the retransmissions.

3. Disabling delayed ACKs. As noted earlier, delayed ACKs can have a poor interaction with microsecond timeouts: the sender may timeout on the last packet in a request, retransmitting it while waiting for a delayed ACK from the receiver. In the datacenter environment, the reduced window size is relatively unimportant—maximum window sizes are small already and the RTT is short. This retransmission does, however, create overhead when requests are small, reducing throughput by roughly 10%.

While the focus of this paper is on understanding the limits of providing high throughput in datacenter networks, it is important to ensure that these solutions can be adopted incrementally. Towards this end, we discuss briefly three mechanisms for interacting with legacy clients:

- *Bimodal timeout operation*: The kernel could use microsecond RTTs only for extremely low RTT connections, where the retransmission overhead is much lower than the cost of coarse-grained timeouts, but still use coarse timeouts in the wide-area.

- *Make aggressive timeouts a socket option*: Require users to explicitly enable aggressive timeouts for their application. We believe this is an appropriate option for the first stages of deployment, while clients are being upgraded.

- *Disable delayed ACKs, or use an adaptive delayed ACK timer.* While the former has numerous advantages, fixing the delay penalty of delayed ACK for the datacenter is relatively straightforward: Base the delayed ACK timeout on the smoothed inter-packet arrival rate instead of having a static timeout value. We have not implemented this option, but as Figure 6 shows, a static $200\mu s$ timeout value (still larger than the inter-packet arrival) shows that these more precise delayed ACKs restore throughput to about 98% of the no-delayed-ACK throughput.

6 Achieving Microsecond-granularity Timeouts

The TCP clock granularity in most popular operating systems is on the order of milliseconds, as defined by the "jiffy" clock, a global counter updated by the kernel at a frequency "HZ", where HZ is typically 100, 250, or 1000. Linux, for example, updates the jiffy timer 250 times a second, yielding a TCP clock granularity of 4ms, with a configuration option to update 1000 times per second for a 1ms granularity. More frequent updates, as would be needed to achieve finer granularity timeouts, would impose an increased, system-wide kernel interrupt overhead.

Unfortunately, setting the RTO_{min} to 1 jiffy (the lowest possible value) does not achieve RTO values of 1ms because of the clock granularity. TCP measures RTTs in 1ms granularity at best, so both the smoothed RTT estimate and RTT variance have a 1 jiffy (1ms) lower bound. Since the standard RTO estimator sums the RTT estimate with 4x the RTT variance, the lowest possible RTO value is 5 jiffies. We experimentally validated this result by setting the clock granularity to 1ms, setting RTO_{min} to 1ms, and observing that TCP timeouts were a minimum of 5ms.

At a minimum possible RTO_{min} of 5ms in standard TCP implementations, throughput loss can not be avoided for as few as 8 concurrent senders. While the results of Figures 3 and 4 suggest reducing the RTO_{min} to 5ms can be both simple (a one-line change) and helpful, next we describe how to achieve microsecond granularity RTO values in the real world.

6.1 Linux high-resolution timers: hrtimers

High resolution timers were introduced in Linux kernel version 2.6.18 and are still actively in development. They form the basis of the posix-timer and itimer user-level timers, nanosleep, and a few other in-kernel operations, including the update of the jiffies value.

The Generic Time of Day (GTOD) framework provides the kernel and other applications with nanosecond resolution timing using the CPU cycle counter on modern processors. The hrtimer implementation interfaces with the High Precision Event

Timer (HPET) hardware also available on modern systems to achieve microsecond resolution in the kernel. When a timer is added to the list, the kernel checks whether this is the next expiring timer, programming the HPET to send an interrupt when the HPET's internal clock advances by a desired amount. For example, the kernel may schedule a timer to expire once every 1ms to update the jiffy counter, and the kernel will be interrupted by the HPET to update the jiffy timer only every 1ms.

Our preliminary evaluations of hrtimer overhead have shown no appreciable overhead of implementing TCP timeouts using the hrtimer subsystem. We posit that, at this stage in development, only a few kernel functions use hrtimers, so the red-black tree that holds the list of timers may not contain enough timers to see poor performance as a result of repeated insertions and deletions. Also, we argue that for incast, where high-resolution timers for TCP are required, any introduced overhead may be acceptable, as it removes the idle periods that prevent the server from doing useful work to begin with.

6.2 Modifications to the TCP Stack

The Linux TCP implementation requires three changes to support microsecond timeouts using hrtimers: microsecond resolution time accounting to track RTTs with greater precision, redefinition of TCP constants, and replacement of low-resolution timers with hrtimers.

Microsecond accounting: By default, the jiffy counter is used for tracking time. To provide microsecond granularity accounting, we use the GTOD framework to access the 64-bit nanosecond resolution hardware clock wherever the jiffies time is traditionally used.

With the TCP timestamp option enabled, RTT estimates are calculated based on the difference between the timestamp option in an earlier packet and the corresponding ACK. We convert the time from nanoseconds to microseconds and store the value in the TCP timestamp option[3]. This change can be accomplished entirely on the sender—receivers simply echo back the value in the TCP timestamp option.

[3]The lower wrap-around time – 2^{32} microseconds or 4294 seconds – is still far greater than the maximum IP segment lifetime (120-255 seconds)

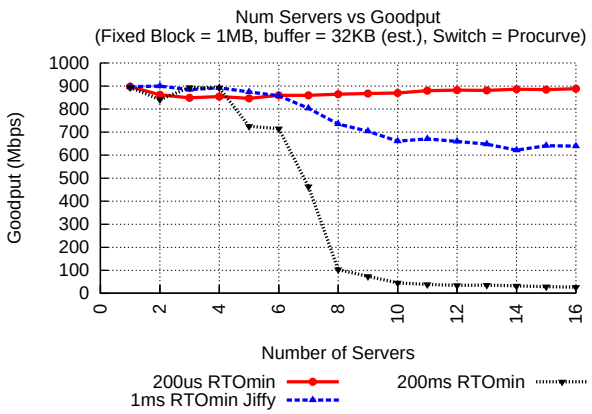


Figure 7: On a 16 node cluster, our high-resolution 1ms RTO_{min} eliminates incast. The jiffy-based implementation has a 5ms lower bound on RTO, and achieves only 65% throughput.

Constants, Timers and Socket Structures All timer constants previously defined with respect to the jiffy timer are converted to exact microsecond values. The TCP implementation must make use of the hrtimer interface: we replace the standard timer objects in the socket structure with the hrtimer structure, ensuring that all subsequent calls to set, reset, or clear these timers use the appropriate hrtimer functions.

6.3 hrtimer Results

Figure 7 presents the achieved goodput as we increase the stripe width N using various RTO_{min} values on a Procurve 2848 switch. As before, the client issues requests for 1MB data blocks striped over N servers, issuing the next request once the previous data block has been received. Using the default 200ms RTO_{min}, throughput plummets beyond 8 concurrent senders. For a 1ms jiffy-based RTO_{min}, throughput begins to drop at 8 servers to about 70% of link capacity and slowly decreases thereafter; as shown previously, the effective RTO_{min} is 5ms. Last, our TCP hrtimer implementation allowing microsecond RTO values achieves the maximum achievable goodput for 16 concurrent senders.

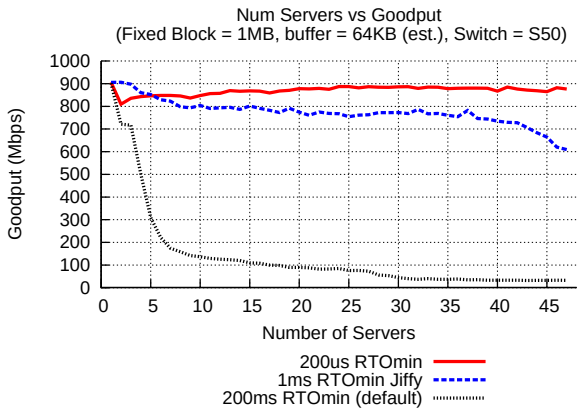


Figure 8: For a 48-node cluster, providing RTO values in microseconds eliminates incast.

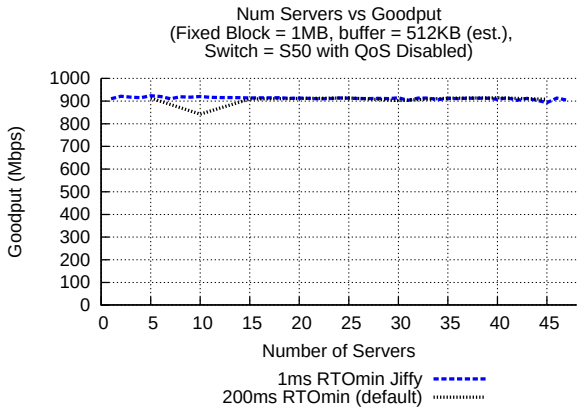


Figure 9: Switches configured with large enough buffer capacity can delay incast.

We verify these results on a second cluster consisting of 1 client and 47 servers connected to a single 48-port Force10 S50 switch (Figure 8). The microsecond RTO kernel is again able to saturate throughput for 47 servers, the testbed's maximum size. The 1ms RTO_{min} jiffy-based configuration obtained 70-80% throughput, with an observable drop above 40 concurrent senders.

When the Force10 S50 switch is configured to disable multiple QoS queues, the per-port packet buffer allocation is large enough that incast can be avoided for up to 47 servers (Figure 9). This reaffirms the simulation result of Figure 2 – that one way to avoid incast in practice is to use larger per-port buffers in switches on the path. It also emphasizes that relying on switch-based incast solutions involves more

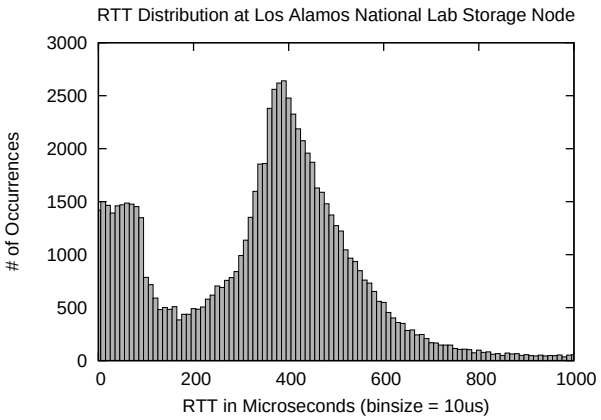


Figure 10: Distribution of RTTs shows an appreciable number of RTTs in the 10s of microseconds.

than just the switches total buffer size: switch configuration options designed for other workloads can make its flows more prone to incast. A generalized TCP solution should reduce administrative complexities in the field.

Overall, we find that enabling microsecond RTO values in TCP successfully avoids the incast throughput drop in two real-world clusters for as high as 47 concurrent servers, the maximum available to us to date, and that microsecond resolution is *necessary* to achieve high performance with some switches or switch configurations.

7 Next-generation Datacenters

TCP Incast poses further problems for the next generation of datacenters consisting of 10Gbps networks and hundreds to thousands of machines. These networks have very low-latency capabilities to keep competitive with alternative high-performance interconnects like Infiniband; though TCP kernel-to-kernel latency may be as high as 40-80μs due to kernel scheduling, port-to-port latency can be as low as 10μs. Because 10Gbps Ethernet provides higher bandwidth, servers will be able to send their portion of a data block faster, requiring that RTO values be shorter to avoid idle link time. For example, we plot the distribution of RTTs seen at a storage node at Los Alamos National Lab (LANL) in Figure 10: 20% of RTTs are below 100μs, showing that networks today are capable and operate in very low-latency environ-

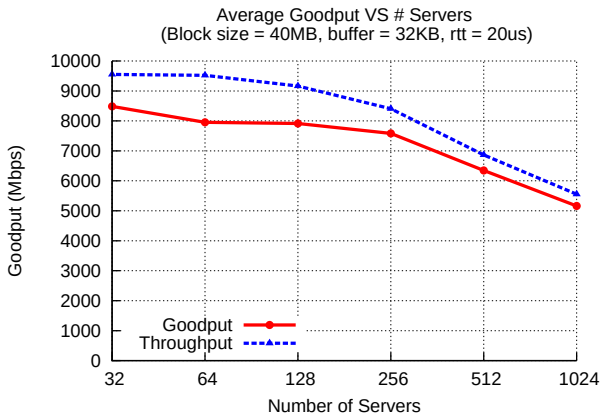


Figure 11: In simulation, flows still experience reduced goodput even with microsecond resolution timeouts (RTO_{min} = RTT = 20μs) without a randomized RTO component.

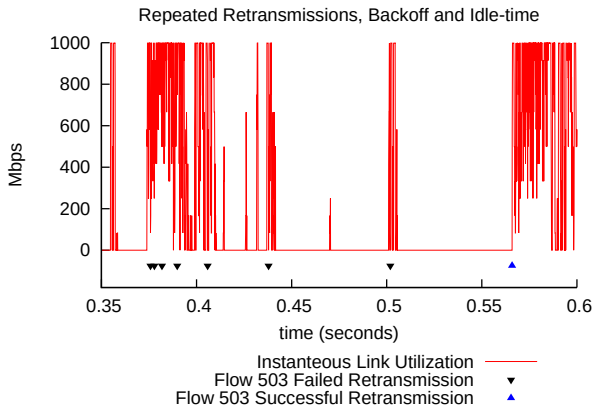


Figure 12: Some flows experience repeated retransmission failures due to synchronized retransmission behavior, delaying transmission far beyond when the link is idle.

ments, further motivating the need for microsecond-granularity RTO values.

Scaling to thousands of concurrent senders: In this section, we analyze the impact of incast and the reduced RTO solution for 10Gbps Ethernet networks in simulation as we scale the number of concurrent senders into the thousands. We reduce baseline RTTs from 100μs to 20μs, and increase link capacity to 10Gbps, keeping per-port buffer size at 32KB as we assume smaller buffers for faster switches.

We increase the blocksize to 40MB (to ensure each flow can mostly saturate the 10Gbps link), scale up the number of nodes from 32 to 2048, and reduce the RTO_{min} to 20μs, effectively eliminating a minimum bound. In Figure 11, we find that without a randomized RTO component, goodput decreases sublinearly (note the log-scale x-axis) as the number of nodes increases, indicating that even with an aggressively RTO granularity, we still observe reduced goodput.

Reduced goodput can arise due to idle link time or due to retransmissions; retransmissions factor into throughput but not goodput. Figure 11 shows that for a small number of flows, throughput is near optimal but goodput is lower, sometimes by up to 15%. For a larger number of flows, however, throughput and goodput are nearly identical – with an aggressively low RTO, there exist periods where the link is idle for a large number of concurrent senders.

These idle periods occur specifically when there are many more flows than the amount of buffer capacity at the switch due to *simultaneous, successive timeouts*. Recall that after every timeout, the RTO value is doubled until an ACK is received. This has been historically safe because TCP quickly and conservatively estimates the duration to wait until congestion abates. However, the exponentially increasing delay can overshoot some portion of time that the link is actually idle, leading to sub-optimal goodput. Because only one flow must overshoot to delay the entire transfer, the probability of overshooting increases with increased number of flows.

Figure 12 shows the instantaneous link utilization for all flows and the retransmission events for one of the flows that experienced repeated retransmission failures during an incast simulation on a 1Gbps network. This flow timed out and retransmitted a packet at the same time that other timed out flows also retransmitted. While some of these flows got through and saturated the link for a brief period of time, the flow shown here timed out and doubled its timeout value (until the maximum factor of 64 * RTO) for each failed retransmission. The link then became available soon after the retransmission event, but the RTO backoff set the retransmission timer to fire far beyond this time. When this packet eventually got through, the block transfer completed and the next block transfer began, but only after large periods of link idle time that reduced goodput.

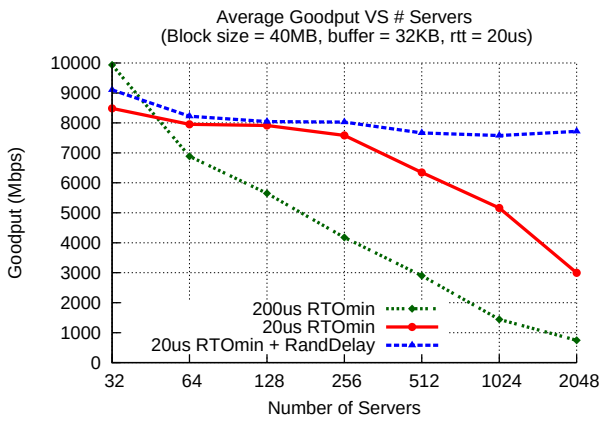


Figure 13: In simulation, introducing a randomized component to the RTO desynchronizes retransmissions following timeouts and avoiding goodput degradation for a large number of flows.

This analysis shows that decreased goodput/throughput for a large number of flows can be attributed to many flows timing out simultaneously, backing off *deterministically*, and retransmitting at precisely the same time. By adding some degree of randomness to the RTO, the retransmissions can be desynchronized such that fewer flows experience repeated timeouts.

We examine both RTO_{min} and the retransmission synchronization effect in simulation, measuring the goodput for three different settings: a 200μs RTO_{min}, a 20μs RTO_{min}, and a 20μs RTO_{min} with a modified, randomized timeout value set by:

$$\text{timeout} = RTO + (rand(0.5) * RTO)$$

Figure 13 shows that the 200μs RTO_{min} scales poorly as the number of concurrent senders increases: at 1024 servers, throughput is still an order of magnitude lower. The 20μs RTO_{min} shows improved performance, but eventually suffers beyond 1024 servers due to the successive, simultaneous timeouts experienced by a majority of flows.

Adding a small random delay performs well regardless of the number of concurrent senders because it explicitly desynchronizes the retransmissions of flows that experience repeated timeouts, and does not heavily penalize flows that experience a few timeouts.

A caveat of these event-based simulations is that the timing of events is as accurate as the simulation timestep. At the scale of microseconds, there will exist small timing and scheduling differences in the real-world that are not captured in simulation. For example, in the simulation, a packet will be retransmitted as soon as the retransmission timer fires, whereas kernel scheduling may delay the actual retransmission by 10μs or more. Even when offloading duties to ASICs, slight timing differences will exist in real-world switches. Hence, the real-world behavior of incast in 10GE, 20μs RTT networks will likely deviate from these simulations slightly, though the general trend should hold.

8 Implications of Reduced RTOmin on the Wide-area

Aggressively lowering both the RTO and RTO_{min} shows practical benefits for datacenters. In this section, we investigate if reducing the RTO_{min} value to microseconds and using finer granularity timers is safe for wide area transfers. We find that the impact of spurious timeouts on long, bulk data flows is very low – within the margins of error – allowing RTO to go into the microseconds without impairing wide-area performance.

8.1 Evaluation

The major potential effect of a spurious timeout is a loss of performance: a flow that experiences a timeout will reduce its slow-start threshold by half, its window to one and attempt to rediscover link capacity. Spurious timeouts occur not when the network path drops packets, but rather when it observes a sudden, higher delay, so the effect of a shorter RTO on increased congestion is likely small because a TCP sender backs-off on the amount of data it injects into the network on a timeout. In this section we analyze the performance of TCP flows over the wide-area for bulk data transfers.

Experimental Setup We deployed two servers that differ only in their implementation of the RTO values and granularity, one using the default Linux 2.6.28 kernel with a 200ms RTO_{min}, and the other using our modified hrtimer-enabled TCP stack with

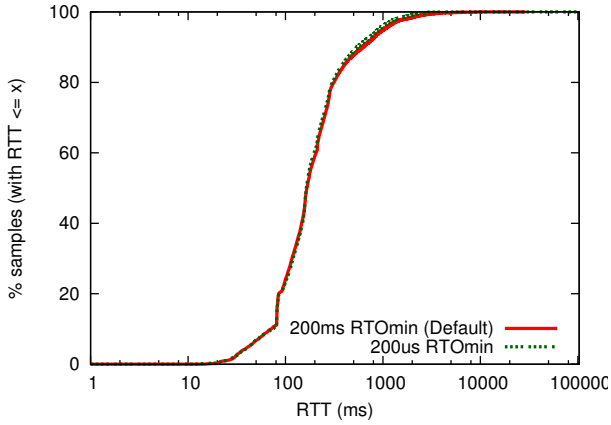


Figure 14: A comparison of RTT distributions of flows collected over 3 days on the two configurations show that both servers saw a similar distribution of both short and long-RTT flows.

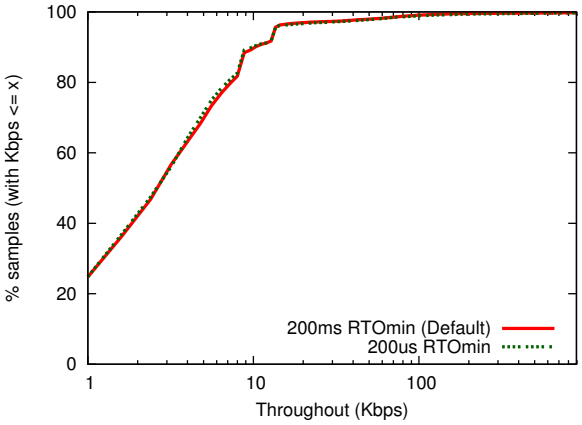


Figure 15: The two configurations observed an identical throughput distribution for flows. Only flows with throughput over 100 bits/s were considered.

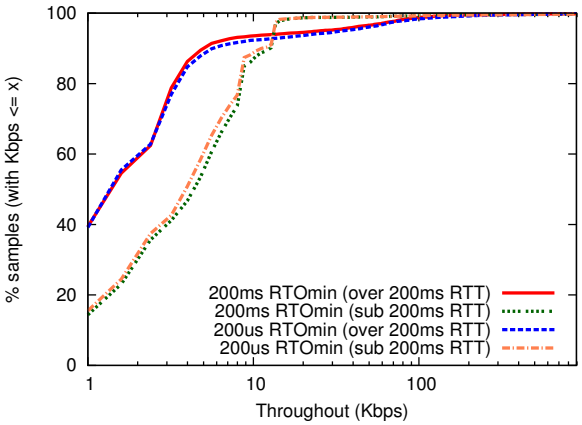


Figure 16: The throughput distribution for short and long RTT flows shows negligible difference across configurations.

a 200μs RTO_{min}. We downloaded 12 torrent files consisting of various Linux distributions and begin seeding all content from both machines on the same popular swarms for three days. Each server uploaded over 30GB of data, and observed around 70,000 flows (with non-zero throughput) over the course of three days. We ran tcpdump on each machine to collect all uploaded traffic packet headers for later analysis.

The TCP RTO value is determined by the estimated RTT value of each flow. Other factors being equal, TCP throughput tends to decrease with increased RTT. To compare RTO and throughput metrics for the 2 servers we first investigate if they see similar flows with respect to RTT values. Figure 14 shows the per-flow average RTT distribution for both hosts over the three day measurement period. The RTT distributions are nearly identical, suggesting that each machine saw a similar distribution of both short and long-RTT flows. The per-packet RTT distribution for both flows are also identical.

Figure 15 shows the per-flow throughput distributions for both hosts, filtering out those flows with a bandwidth less than 100bps, which are typically flows sending small control packets. The throughput distributions for both hosts are also nearly identical – the host with RTO_{min} = 200μs did not perform worse on the whole than the host with RTO_{min} = 200ms.

We split the throughput distributions based on whether the flow's RTT was above or below 200ms. For flows above 200ms, we use the variance in the two distributions as a control parameter: any variance seen above 200ms are a result of measurement noise, because the RTO_{min} is no longer a factor. Figure 16 shows that the difference between the distribution for flows below 200ms is within this measurement noise.

Table 1 lists statistics for the number of spurious timeouts and normal timeouts observed by both the servers. These statistics were collected using tcptrace [2] patched to detect timeout events [1]. The number of spurious timeouts observed by the two configurations are high, but comparable. We at-

RTO_{min}	Normal timeouts	Spurious timeouts	Spurious Fraction
200ms	137414	47094	25.52%
200μs	264726	90381	25.45%

Table 1: Statistics for timeout events across flows for the 2 servers with different RTO_{min} values. Both servers experience a similar % of spurious timeouts.

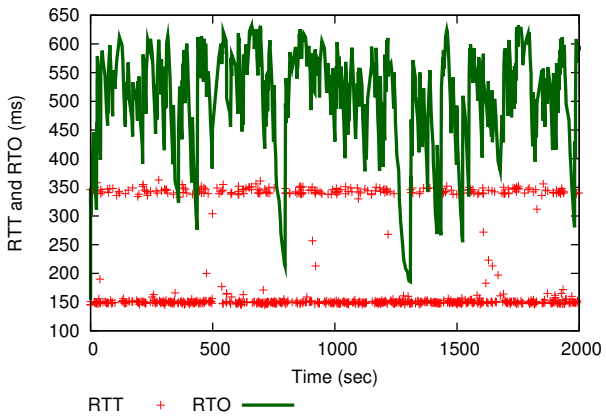


Figure 17: RTT and RTO estimate over time for a randomly picked flow over a 2000 second interval – the RTT and RTO estimate varies significantly in the wide-area which may cause spurious timeouts.

tribute the high number of spurious timeouts to the nature of TCP clients in our experimental setup. The peers in our BitTorrent swarm observe large variations in their RTTs (2x-3x) due to the fact that they are transferring large amounts of data, frequently establishing and breaking TCP connections to other peers in the swarm, resulting in variations in buffer occupancy of the bottleneck link.[4] Figure 17 shows one such flow picked at random, and the per-packet RTT and estimated RTO value over time. The dip in the estimated RTO value below the 350ms RTT band could result in a spurious timeout if there were a timeout at that instance and the instantaneous RTT was 350ms.

This data suggests that reducing the RTO_{min} to 200μs in practice does not affect the performance of bulk-data TCP flows on the wide-area.

[4]An experiment run by one of the authors discovered that RTTs over a residential DSL line varied from 66ms for one TCP stream to two seconds for four TCP streams.

9 Related Work

TCP Improvements: A number of TCP improvements over the years have improved TCP's ability to respond to loss patterns and perform better in particular environments, many of which are relevant to the high-performance datacenter environment we study. NewReno and SACK, for instance, reduce the number of loss patterns that will cause timeouts; prior work on the incast problem showed that NewReno, in particular, improved throughput during moderate amounts of incast, though not when the problem became severe [28].

TCP mechanisms such as Limited Transmit [3] were specifically designed to help TCP recover from packet loss when window sizes are small—exactly the problem that occurs during incast. This solution again helps maintain throughput under modest congestion, but during severe incast, the most common loss pattern is the loss of the entire window.

Finally, proposed improvements to TCP such as TCP Vegas [10] and FAST TCP [19] can limit window growth when RTTs begin to increase, often combined with more aggressive window growth algorithms to rapidly fill high bandwidth-delay links. Unlike the self-interfering oscillatory behavior on high-BDP links that this prior work seeks to resolve, Incast is triggered by the arrival and rapid ramp-up of numerous competing flows, and the RTT increases drastically (or becomes a full window loss) over a single round-trip. While an RTT-based approach is an interesting approach to study for alternative solutions to Incast, it is a matter of considerable future work to adapt existing techniques for this purpose.

Efficient, fine-grained kernel timers. Where our work depends on hardware support for high-resolution kernel timers, earlier work on "soft timers" shows an implementation path for legacy systems [5]. Soft timers can provide microsecond-resolution timers for networking without introducing the overhead of context switches and interrupts. The hrtimer implementation we do make use of draws lessons from soft timers, using a hardware interrupt to trigger all available software interrupts.

Understanding RTO_{min}. The origin of concern about the safety and generality of reducing RTO_{min} was presented by Allman and Paxson [4], where they used trace-based analysis to show that there existed

no optimal RTO estimator, and to what degree that the TCP granularity and RTO_{min} had an impact on spurious retransmissions. Their analysis showed that a low or non-existent RTO_{min} greatly increased the chance of spurious retransmissions and that tweaking the RTO_{min} had no obvious sweet-spot for balancing fast response with spurious timeouts. They showed the increased benefit of having a fine measurement granularity for responding to good timeouts because of the ability to respond to minor changes in RTT. Last, they suggested that the impact of bad timeouts could be mitigated by using the TCP timestamp option, which later became known as the Eifel algorithm [21]. F-RTO later showed how to detect spurious timeouts by detecting whether the following acknowledgements were for segments not retransmitted [32], and this algorithm is implemented in Linux TCP today.

Psaras and Tsaoussidis revisit the minimum RTO for high-speed, last-mile wireless links, noting the default RTO_{min} is responsible for worse throughput on wireless links and short flows [29]. They suggest a mechanism for dealing with delayed ACKs that attempts to predict when a packet's ACK is delayed – a per-packet RTO_{min}. We find that while delayed ACK can affect performance for low RTO_{min}, the benefits of a low RTO_{min} far outweigh the impact of delayed ACK on performance. Regardless, we provide alternative, backwards-compatible solutions for dealing with delayed ACK.

10 Conclusion

This paper presented a *practical*, *effective*, and *safe* solution to eliminate TCP throughput degradation in datacenter environments. Using a combination of microsecond granularity timeouts, randomized retransmission timers, and disabling delayed acknowledgements, the techniques in this paper allowed high-fan-in datacenter communication to scale to 47 nodes in a real cluster evaluation, and potentially to thousands of nodes in simulation. Through a wide-area evaluation, we showed that these modifications remain safe for use in the wide-area, providing a general and effective improvement for TCP-based cluster communication.

References

[1] TCP Spurious Timeout detection patch for tcptrace. http://ccr.sigcomm.org/online/?q=node/220.

[2] tcptrace: A TCP Connection Analysis Tool. http://irg.cs.ohiou.edu/software/tcptrace/.

[3] M. Allman, H. Balakrishnan, and S. Floyd. *Enhancing TCP's Loss Recovery Using Limited Transmit*. Internet Engineering Task Force, January 2001. RFC 3042.

[4] Mark Allman and Vern Paxson. On estimating end-to-end network path properties. In *Proc. ACM SIGCOMM*, Cambridge, MA, September 1999.

[5] Mohit Aron and Peter Druschel. Soft timers: Efficient Microsecond Software Timer Support for Network Processing. *ACM Transactions on Computer Systems*, 18(3):197–228, 2000.

[6] H. Balakrishnan, V. N. Padmanabhan, and R.H. Katz. The Effects of Asymmetry on TCP Performance. In *Proc. ACM MOBICOM*, Budapest, Hungary, September 1997.

[7] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R.H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. In *Proc. ACM SIGCOMM*, Stanford, CA, August 1996.

[8] Peter J. Braam. File Systems for Clusters from a Protocol Perspective. http://www.lustre.org.

[9] R. T. Braden. *Requirements for Internet Hosts—Communication Layers*. Internet Engineering Task Force, October 1989. RFC 1122.

[10] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proc. ACM SIGCOMM*, London, England, August 1994.

[11] k. claffy, G. Polyzos, and H-W. Braun. Measurement considerations for assessing unidirectional latencies. *Internetworking: Research and Experience*, 3(4):121–132, September 1993.

[12] Scaling memcached at Facebook. http://www.facebook.com/note.php?note_id=39391378919.

[13] Brad Fitzpatrick. LiveJournal's backend: A history of scaling. Presentation, http://tinyurl.com/67s363, August 2005.

[14] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4), August 1993.

[15] Bryan Ford. Structured streams: A new transport abstraction. In *Proc. ACM SIGCOMM*, Kyoto, Japan, August 2007.

[16] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, October 2003.

[17] V. Jacobson. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM*, pages 314–329, Vancouver, British Columbia, Canada, September 1998.

[18] V. Jacobson, R. Braden, and D. Borman. *TCP Extensions for High Performance*. Internet Engineering Task Force, May 1992. RFC 1323.

[19] Cheng Jin, David X. Wei, and Steven H. Low. FAST TCP: motivation, architecture, algorithms, performance.

[20] Eddie Kohler, Mark Handley, and Sally Floyd. Designing DCCP: Congestion control without reliability. In *Proc. ACM SIGCOMM*, Pisa, Italy, August 2006.

[21] R. Ludwig and M. Meyer. *The Eifel Detection Algorithm for TCP*. Internet Engineering Task Force, April 2003. RFC 3522.

[22] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. *TCP Selective Acknowledgment Options*. Internet Engineering Task Force, 1996. RFC 2018.

[23] Alberto Medina, Mark Allman, and Sally Floyd. Measuring the evolution of transport protocols in the Internet. April 2005.

[24] Amarnath Mukherjee. On the dynamics and significance of low frequencey components of internet load. *Internetworking: Research and Experience*, 5:163–205, December 1994.

[25] David Nagle, Denis Serenyi, and Abbie Matthews. The Panasas ActiveScale Storage Cluster: Delivering Scalable High Bandwidth Storage. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, Washington, DC, USA, 2004.

[26] ns-2 Network Simulator. http://www.isi.edu/nsnam/ns/, 2000.

[27] C. Partridge. *Gigabit Networking*. Addison-Wesley, Reading, MA, 1994.

[28] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *Proc. USENIX Conference on File and Storage Technologies*, San Jose, CA, February 2008.

[29] Ioannis Psaras and Vassilis Tsaoussidis. The tcp minimum rto revisited. In *IFIP Networking*, May 2007.

[30] K. Ramakrishnan and S. Floyd. *A Proposal to Add Explicit Congestion Notification (ECN) to IP*. Internet Engineering Task Force, January 1999. RFC 2481.

[31] S. Raman, H. Balakrishnan, and M. Srinivasan. An Image Transport Protocol for the Internet. In *Proc. International Conference on Network Protocols*, Osaka, Japan, November 2000.

[32] P. Sarolahti and M. Kojo. *Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP and the Stream Control Transmission Protocol (SCTP)*. Internet Engineering Task Force, August 2005. RFC 4138.

[33] S. Shepler, M. Eisler, and D. Noveck. NFSv4 Minor Version 1 – Draft Standard.

[34] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jim Zelenka, and Bin Zhou. Scalable Performance of the Panasas Parallel File System. In *Proc. USENIX Conference on File and Storage Technologies*, San Jose, CA, February 2008.

[35] Y. Zhang, N. Duffield, V. Paxson, and S. Shenker. On the constancy of Internet path properties. In *Proc. ACM SIGCOMM Internet Measurement Workshop*, San Fransisco, CA, November 2001.